УДК 681.3.06

В.И. Гриценко, А.В. Анисимов, Н.Д. Пашковец, О.В. Бабак

Автоматизация построения бизнес-правил в процессе реинжиниринга программных *legacy*-систем на этапе анализа их функциональных структур

Предлагаемая статья представляет собой логическое продолжение материала, изложенного в статье «Реализация реинжиниринга программных *legacy*-систем». В данной статье изложено обоснование подхода к реинжинирингу программных *legacy*-систем и определяются условия корректного построения комплекса бизнес-правил, а также описаны алгоритмы построения и методика их проверки.

The article is a logic continuation of the material stated in the article «Realization of reengineering of program legacy-systems». In the present article the substantiation of the approach to reengineering of the program legacy-systems is stated and the conditions of the correct construction of a complex of business-rules are defined, as well as the algorithms of construction and a technique of their check are described.

Пропонована стаття є логічним продовженням матеріалу, викладеного в статті «Реализация реинжиниринга программных *legacy*-систем». В даній статті подано обґрунтування запропонованого підходу до реінжинірингу програмних *legacy*-систем і визначаються умови коректної побудови комплексу бізнес-правил, а також описано алгоритми побудови і методика їх перевірки.

Введение. В работе [1] определено, что процесс реинжиниринга *legacy*-систем можно разбить на три этапа:

- разбор функциональной структуры *legacy*-системы (стадия *Preprocessing*);
- анализ модулей программной системы на предмет определения бизнес-терминов, выделения и поиска паттернов с целью написания бизнес-правил (БП);
- моделирование программной *legacy*-системы непосредственно через найденные БП с целью изменения архитектуры программной системы с учетом новых требований к системе, а также особенностей применяемых технологических и информационных платформ.

В статье изложено обоснование предлагаемого подхода к реинжинирингу программных *legacy*-систем и описаны алгоритмы, относящиеся к первому этапу, ориентированные на автоматизацию построения БП на стадии *Preprocessing*.

В [1] была положена в основу концепция, что всякий программный модуль, независимо от технологии его разработки, в конечном счете состоит из следующих основных компонент: линейных участков программ, процессов ветвления, процессов вложенности и циклических процессов. Для формального описания этих конструкций было введено понятие статических и динамических векторов состояния, соответствующих наборам переменных, связанным с конкретным БП.

Постановка задачи

С учетом изложенного поставлена задача построения конкретных методов и механизмов автоматического фиксирования и отслеживания состояния логических переменных, связанных с предметной областью, с целью организации последовательности и контроля вызова БП, а также определение условий корректного построения комплекса БП и разработка методики их проверки при проведении реинжиниринга программных *legacy*-систем.

Решение задачи. Правила описания и построения векторов состояния

В процессе написания конкретного БП, как составляющего комплекса БП, для него определяется набор логических условий, определенные значения которых вызывают активизацию данного БП. Опыт добывания БП из программного кода показывает, что из множества флажков, активизирующих БП, как правило, один флажок выполняет функцию его начального включения. Назовем такие флажки (логи-

ческие условия) *первичными*. Например, к первичным можно отнести флажки-сцепления и объявляемые логические переменные.

Всякий флажок в плане начальной активизации (объявления), как правило, связан с определенным БП, но может использоваться во многих других. В первом случае флажок адекватен первичному, во втором – выполняет роль *дополнительных* условий. Поэтому все множество формируемых флажков комплекса БП целесообразно представить в виде матрицы размерностью $(m + n, m)$, где m – количество БП, $m + n$ – количество используемых в них флажков, а n – количество дополнительных условий. Подматрица (m, m) предназначена для фиксирования и отслеживания использования первичных флажков, причем в качестве как первичных, так и дополнительных условий. В подматрице (n, m) фиксируется и отслеживается использование исключительно дополнительных условий.

Этот принцип отображения множества флажков на множество БП посредством матричного представления используем для построения статических векторов состояния, где столбцы матрицы будут отвечать векторам условий активизации соответствующих БП, а сама матрица представляет собой набор условий запуска БП. Рабочая матрица условий запуска БП может быть построена автоматически путем просмотра всех БП и выборки из них логических переменных (флажков) и условий запуска БП.

При написании БП целесообразно сначала вручную строить матрицу условий запуска БП, расширяя ее по мере добавления БП и флажков, что дает возможность следить за формированием условий активизации создаваемых БП в плане соблюдения их уникальности. Этот процесс поддается автоматизации с выполнением некоторой оптимизации относительно вводимых логических переменных в процессе написания БП, а также для проверки непротиворечивости запуска БП, что значительно повысит производительность и качество написания БП.

После построения матрицы условий запуска БП создается статический вектор состояния флажков размерностью $m + n$, в котором будет фиксироваться текущее состояние всех логических переменных комплекса БП. Если комплекс БП построен корректно, то для всякой конфигурации вектора состояния найдется один и только один адекватный ему столбец в матрице условий запуска БП, что и вызовет запуск соответствующего БП.

Дадим определение корректного построения комплекса БП.

Будем считать, что комплекс БП построен корректно, если входящие в его состав БП удовлетворяют условиям: непротиворечивости, полноты, необходимости и достаточности.

БП *непротиворечивы*, если все столбцы матрицы условий запуска БП уникальны.

БП *достаточны*, если для всякой текущей конфигурации вектора состояния имеется адекватный столбец в матрице условий запуска БП.

БП *необходимо*, если столбец в матрице условий запуска БП активизировался хотя бы один раз.

Полнота БП в данном случае вытекает из выполнения условий необходимости и достаточности.

Технология фиксирования и отслеживания состояния динамических объектов, описанных выше, осуществляется аналогично статическим, с тем отличием, что матрица и вектор состояния имеют динамический характер.

Из изложенного видно, что для полустатических и динамических БП [1] условие активизации является двухуровневым: на первом уровне проверяется статическая конфигурация вектора состояния, на втором – динамическая. Выполнение обоих условий вызывает активизацию БП.

Для унификации активизации БП различных видов можно ввести отдельный вектор активизации правил, представляющий собой бинарный вектор, размер которого равен количеству БП. В каждый момент времени в векторе активизации «включен» только один элемент, соответствующий по индексу активному БП. В конце отработки БП этот элемент «выключается», а элемент, соответствующий активизи-

руемому БП включается. Таким образом, вектор активизации отражает порядок и время выполнения БП, что позволит организовать обработку очередей для параллельных процессов и реализацию мультизадачного режима.

С целью облегчения ручного формирования матрицы условий запуска БП рекомендуется следующий формат объявления логических переменных:

F(*I*):<имя флажка> – для первичных флажков, где буква *F* – признак первичности флажка, а индекс *I* – номер БП (имеет диапазон изменения от единицы до *m*).

S(*J*): <имя флажка> – для дополнительных флажков, где буква *S* – признак дополнительного (вторичного) условия, а индекс *J* имеет диапазон изменения от *m* + 1 до *n*.

Примеры: *F*(4):*Open file* – объявление первичной логической переменной;

S(36):*Move data* – объявление дополнительной логической переменной.

Связь с предметной областью

Изложенный материал касается в основном организации вычислительного процесса. Рассмотрим вопросы связи вычислительного процесса с предметной областью, а именно увязку БП, работа которых в основном базируется на обработке логических выражений, с реальными программными объектами. Такая связь осуществляется через механизм *Action* (БП-действия) [2, 3]. В связи с тем, что изложенный механизм построения БП не привязывался к языку программирования, БП-действия можно оформлять в виде отдельных программных блоков на требуемом языке программирования и с использованием различных технологий, в том числе и объектно-ориентированным. Вероятно, возможно разрешить для БП-действий вложенность, т.е. БП-действие – это не только линейная последовательность каких-либо действий, но и, в свою очередь, программный объект со своими БП.

БП-действия формируются и накапливаются в процессе написания БП. Языком описания БП-действия может быть либо конкретный язык программирования, либо какой-то метаязык, с которого в нужный язык программирования осуществляется перевод с помощью программных средств, в простейшем виде представляющих собой набор соответствующих парсеров.

Для удобства связывания БП и БП-действий рекомендуется следующий формат объявления БП-действия:

A(*I*):<имя действия>,

где буква *A* – признак действия, индекс *I* – номер БП (диапазон изменений от единицы до *m*).

Конкретное БП-действие может использоваться более чем в одном БП. Поэтому в репозитории БП-действия необходимо хранить в виде реляционной таблицы, подобной описанной матрице условий запуска БП.

Заметим, что проблема представления БП-действий, от которой существенно зависит эффективность генерируемого выходного кода очень важна, поэтому, безусловно, является предметом отдельного исследования. Кроме того, именно через представления БП-действий можно решить вопросы интеграции генерируемого кода в различные внешние системы или подключать внутренние прикладные задачи.

Общее описание операций, выполняемых на стадии *Preprocessing*

Разбор функциональной структуры программной *legacy*-системы предполагает выполнение следующих работ.

Построение дерева вызовов модулей системы с целью выявления активной ее части, установления порядка и условий вызовов модулей и отсечения неработающей (мертвой) части системы. При этом предполагается выполнение следующих действий:

- проведение лексического анализа программных модулей системы и создание таблиц лексем, соответствующим переменным (построение таблицы переменных модуля);
- построение дерева вызовов в виде матрицы, строки которой содержат имена модулей с перечнем имен всех программ и подпрограмм, к которым имеется обращение из данного модуля.

Определение межмодульных связей с целью построения схемы внутрисистемных и

межсистемных информационных потоков путем выявления и занесения входных и выходных данных для каждого модуля в таблицу переменных модуля. Анализом наследования отмеченных внешних для модуля данных определяются межмодульные связи в анализируемой системе. Выполняется настройка парсера определения межмодульных связей на распознавание тех синтаксических конструкций, которые реализуют прямую или косвенную передачу данных между модулями (операторы вызова программ, общие области данных, области связи и др.) с целью определения имен передаваемых данных и последующей их идентификацией в таблице переменных модуля. На основании отмеченной в таблице переменных модуля информации строится таблица межмодульных связей, в которой отображается межмодульная передача данных, рассматриваемых как предполагаемые кандидаты в бизнес-объекты.

Отслеживание связей наследования между переменными в рамках отдельных модулей заключается в построении цепочек наследования переменных, связанных между собой общностью пересылаемых данных. Эти цепочки идентифицируются именами, которые рассматриваются как кандидаты в бизнес-термины. Идентификация цепочек наследования переменных выполняется по следующим правилам:

- проверяются первый и последний элементы цепочки наследования на предмет принадлежности их к входным и выходным данным соответственно. Проверка выполняется на основе информации из таблицы переменных модуля;
- если в цепочке наследования имеется входное данное, то кандидату в бизнес-объект, соответствующему этой цепочке наследования, будет присвоено имя входной переменной;
- если в цепочке наследования отсутствует входное данное, но имеется выходное данное, то кандидату в бизнес-объект, соответствующему этой цепочке наследования, будет присвоено имя выходной переменной;
- если в цепочке наследования отсутствуют входное и выходное данные, то кандидату в бизнес-объект, соответствующему этой цепочке наследования, будет присвоено имя первой переменной в цепочке, если она порождена с участием каких-либо входных данных. В противном случае предполагается, что цепочка не связана с бизнес-объектами и исключается из рассмотрения.

Заметим, что цепочка наследования может состоять из одной переменной, если в модуле из нее не происходит никакой пересылки данных в другие переменные. Эта переменная может участвовать в порождении другой переменной или вообще быть пассивной.

Присвоенные имена кандидатов в бизнес-термины с выделенными цепочками наследования заносятся в таблицу имен наследования модуля.

Таблица предполагаемых бизнес-терминов каждого модуля прилагается к соответствующему модулю при передаче его аналитику для анализа, и должна послужить для координации проводимого анализа в плане определения списка бизнес-терминов.

Алгоритм построения дерева вызова модулей *legacy*-системы

Результатом лексического анализа есть выделение синтаксических единиц, называемых лексемами, для каждого программного модуля системы. При этом генерируются таблицы, содержащие все обнаруженные лексемы и их типы. Поскольку для выполнения задачи идентификации бизнес-объектов и бизнес-терминов в первую очередь нужны переменные и смысл, вкладываемый в них, выполняется первый проход синтаксического анализатора с целью выделения лексем, соответствующих программным переменным. После этого из сгенерированных лексическим анализатором таблиц лексем отбираются лексемы, соответствующие переменным, вследствие чего получаются таблицы переменных модуля.

Далее выполняется построение дерева вызовов модулей системы, результатом чего является формирование матрицы, каждая строка которой поставлена в соответствие определенному модулю анализируемой системы. Первый

столбец содержит имя этого модуля, а последующие – перечень имен всех программ и подпрограмм, к которым имеется обращение из данного модуля.

На этапе лексического анализа проверяется цепочка символов, представляющая исходный текст анализируемого модуля, с целью распознавания лексем обращения к вызываемым программам.

Для языков программирования *Assembler*, *COBOL*, *PL*/1, *Fortran* это лексема *CALL*. В языке *Assembler* имеется лексема *FETCH*, по которой программный модуль загружается в память с последующей передачей ему управления. Для современных языков программирования, например семейство *C*-языков, обращение к подпрограммам выполняется через механизм встроенных функций, распознавание которых возможно в плане разбора соответствующих синтаксических конструкций, а именно, путем распознавания имен встроенных функций исходя из их объявления в декларативной части программы. В этом случае имена встроенных функций будут интерпретироваться аналогично лексемам обращения к программам. Следовательно, необходимо предусмотреть настройку анализатора на распознавание лексем в синтаксических конструкциях обращения к программам, присущим языку программирования анализируемого модуля. Выполнить это можно созданием специальной таблицы настройки анализатора, в которой перечисляются поисковые образы лексем и/или имена встроенных функций вызова программ для конкретного языка программирования. Информацию в таблицу настройки записывает анализатор в начале своей работы, извлекая эту информацию из набора настройки на входную грамматику.

В начале работы лексический анализатор выделяет поисковые образы, заданные в таблице, и преобразует таблицу в индексируемый массив, размер которого будет определяться количеством поисковых образов. В процессе лексического разбора анализатор будет сканировать по индексу поисковых образов с целью распознания их во входной цепочке символов анализируемой программы, а синтаксический анализатор будет определять принадлежность ее к соответствующей синтаксической конструкции.

После распознавания конструкций вызова программ синтаксический анализатор выделяет имена программ, вызываемых из данного модуля. Для лексемы *CALL* это имя (в кавычках или без кавычек – для некоторых языков), стоящее после лексемы вызова. В ряде языков допустимы косвенные вызовы программ, суть которых заключается в том, что после лексемы *CALL* стоит не имя вызываемой программы в форме литерала, а имя переменной, в качестве значения которой в декларативном операторе программы в процессе инициализации присваивается имя вызываемой программы, представленное в литеральной форме. В этом случае имя после лексемы *CALL* будет без литеральных кавычек. Косвенный вызов программы в языке *COBOL* показан в примере:

01 *ALFA PICTURE X*(4) *VALUE* ‘*BETA*’.

. . .

CALL ALFA USING PAR.

В приведенном примере программа *BETA* вызывается через имя программной переменной *ALFA*.

Сложнее, когда косвенный вызов программ организован в динамическом режиме, т.е. значение переменной *ALFA* изменяется в процессе выполнения программы. В этом случае требуется просмотреть цепочку наследования переменной *ALFA* (см. далее описание алгоритма отслеживания связей наследования между логическими переменными). Все элементы цепочки наследования переменной *ALFA* заносятся в дерево вызова модулей как имена вызываемых модулей.

В некоторых языках (например, *FORTRAN*) допустимо указание после лексемы *CALL* не только имени программы (основная точка входа в модуль), но и имена дополнительных точек входа. Установить этот факт при анализе вызывающего модуля (без дополнительного анализа вызываемого модуля) не представляется возможным. Поэтому в дереве вызова будут

наблюдаться указания на все точки входа вызываемых модулей, к которым были обращения из вызывающих модулей.

Найденные в анализируемом модуле имена вызываемых программ заносятся в строку матрицы, соответствующей этому модулю.

Перечисленные операции выполняются для каждого модуля анализируемой системы. Таким способом строится таблица вызываемых модулей, представляющая дерево вызова модулей в матричном отображении. Таблица выдается на печать в удобной для аналитика форме. Кроме того, информация из файла, в который парсер заносит таблицу вызываемых модулей, должна быть загружена в репозиторий.

Целесообразно на основе таблицы вызываемых модулей сформировать и выдать цепочки вызываемых модулей системы, т.е. все ветки дерева вызовов должны быть представлены в виде последовательности вызываемых модулей с описанием, по возможности, условий их выполнения. Такое представление последовательностей вызываемых модулей полезно для аналитика при выполнении анализа программной системы.

Путем сопоставления исходного списка модулей системы со списком модулей, попавших в таблицу вызываемых модулей, можно определить перечень модулей, по тем или иным причинам не участвующих в функционировании системы. Эти модули, в силу проводившихся модификаций системы в процессе ее эксплуатации, могут оказаться случайными (лишними) в системе, т.е. «мёртвым» кодом на внешнемодульном уровне. В этом случае они удаляются из системы и не подвергаются дальнейшему анализу и реинжинирингу.

Алгоритм определения межмодульных связей

Построение межмодульных связей предполагает связывание между собой всех таблиц переменных модулей, построенных отдельно для каждого модуля анализируемой системы.

На этапе синтаксического анализа происходит распознавание синтаксических конструкций, реализующих прямую или косвенную передачу данных между модулями через параметры в операторах вызова программ, через общие области данных или системные области связи.

Параметры следуют после лексем вызова программ, распознаваемых на этапе лексического анализа. Например, в языке *COBOL* в операторе *CALL* параметры помещаются после лексемы *USING*.

Одновременно с распознаванием имен вызываемых модулей выделяются имена передаваемых данных, заносимых в таблицу передаваемых параметров, где в начало строки помещаются имена вызывающей и вызываемой программ, за которыми следуют имена передаваемых параметров.

Данные, принимаемые вызываемыми модулями от вызывающих модулей помещаются в *общие области данных*. На этапе синтаксического анализа распознаются синтаксические конструкции, описывающие общие области и их данные в анализируемом модуле. Для языка *COBOL* это *LINKAGE SECTION*, для языка *FORTRAN* – оператор *COMMON*, для языка *Assembler* – команда *COM* и т.д. Имена данных из общих областей анализируемого модуля помещаются в соответствующие модулю строки таблицы принимаемых аргументов.

Некоторые данные, например, текущая или системная дата, время, географические координаты и другие – могут извлекаться программным модулем из *системных областей связи*. Как правило, имена данных из системных областей связи – зарезервированные или специальные. Например, в языке *COBOL* это специальные регистры: *CURRENT-DATE* (текущая дата), *TIME-OF-DAY* (текущее время), *COM-REG* (общая область для оператора *MOVE*). Соответствующие им лексемы распознаются на этапе лексического анализа и помещаются в качестве родителя в таблицу имен наследования с целью выявления в дальнейшем их потомков.

Для пары программных модулей (вызывающего и вызываемого), на основании информации из таблицы параметров и таблицы аргументов, ставятся в соответствие между собой переменные и строятся цепочки наследования

передаваемых между ними данных, с привязкой к соответствующим им бизнес-терминам, занесенным в таблицы имен наследования этих модулей.

Цепочки передаваемых между модулями данных (исключая внутримодульные цепочки наследования) вносят в таблицу межмодульных связей, которая помещается в репозиторий, доступна для всех аналитиков и служит упорядочению использования бизнес-терминов, относящихся к области межмодульных связей. Структура таблицы межмодульных связей аналогична структуре таблицы имен наследования, с той разницей, что цепочки в таблице имен наследования отражают продвижение бизнес-объектов по переменным внутри программных модулей, а цепочки в таблице межмодульных связей – продвижение бизнес-объектов между модулями.

При построении цепочек в таблице межмодульных связей может возникнуть несовпадение имен, соответствующих одному и тому же бизнес-объекту (несовпадение бизнес-терминов), в вызываемом и вызывающем модулях, так как они определялись в каждом модуле автономно. В этом случае в качестве обобщенного имени для всей межмодульной цепочки (обобщенный бизнес-термин для всей программной системы) принимается бизнес-термин родителя цепочки. Все остальные бизнес-термины цепочки будут выполнять функции алиасов обобщенного бизнес-термина.

Такой подход позволит обобщить все бизнес-термины и построить семантическое (смысловое, в понятиях предметной области) дерево, что существенно облегчит аналитику понимание логики работы как отдельных программ, так и программой системы в целом.

Алгоритм отслеживания связей наследования между переменными в рамках отдельных модулей

На этом шаге анализа выделяются операторы передачи данных между переменными (анализ наследования переменных).

На этапе синтаксического анализа идентифицируются операции пересылки–присвоения, характерные для языка программирования анализируемого модуля. В качестве входных параметров для данного шага синтаксического анализа необходимо задать все ключевые слова или синтаксические конструкции, связанные с инициированием операций пересылки–присвоения данных между переменными. Например, для языка *COBOL* это лексема *MOVE* или лексема = в операторе *COMPUTE*, а также оператор *ACCEPT*. Для языков *PL*/1, *FORTRAN* и других это лексема ’=’.

При распознавании лексем пересылки-присвоения происходит уточнение характера присвоения, которые могут иметь следующий вид:

- простое присвоение, когда значение одной переменной присваивается другой переменной, является наследуемым;
- присвоение с преобразованием значения исходной переменной; в этом случае получаемое значение является результатом какого-либо преобразования значения одной или нескольких исходных переменных. Указанная переменная называется *порождаемой*, а такой характер присвоения *не является* наследуемым.

Наследуемые переменные заносятся в таблицу имен наследования согласно иерархии (родители → дети → внуки и т.д.). Порождаемые переменные заносятся в таблицу имен наследования как родители, с целью выявления в дальнейшем их потомков. В качестве родителей в таблицу имен наследования заносим и вновь выявленные имена переменных. Таблица имен наследования строится в виде матрицы, первая колонка которой – родитель, вторая – дитя, третья – внук и т.д. В связи с тем, что заранее не известно максимальное количество поколений потомков, физическая реализация таблицы имен наследования выполняется в виде списков.

При пересылке данного в несколько других переменных для каждого принимающего переменного дублируется вся предыдущая часть ветки наследования вплоть до родителя. Аналогично, если в какую-либо переменную пересылаются данные из разных источников, то автоматически дублируется часть ветки от принимающего имени и до последнего потом-

ка. Поэтому деревья наследования в таблице имен наследования будут представлены как множества отдельных ветвей (цепочек) наследования от родителя (корня) и до последнего потомка на ветке.

По окончании анализа наследования переменные, не имеющие потомков и не являющиеся входными или выходными данными, удаляются из таблицы имен наследования. Потому в таблице после выполнения этого этапа останутся только цепочки наследования переменных модуля, связанные с входными и/или выходными данными.

Выделяются цепочки наследования, имеющие в качестве родителей входные данные, и цепочки наследования, имеющие в качестве последнего потомка выходные данные. Цепочки наследования, не содержащие ни входных, ни выходных данных, рассматриваются как цепочки рабочих полей, удаляются из таблицы имен наследования и не связываются с бизнес-терминами.

При занесении переменной в цепочку наследования, необходимо выполнить проверку на статус переменной, т.е. установить, что переменная не является именем составной структуры, именем переопределения (*REDEFINES*) или именем специального уровня (например, для языка *COBOL* это уровень 88, присваивающий имена отдельным значениям переменной).

Если переменные, участвующие в операции пересылки-присвоения, являются именами структур, то необходимо выделить в соответствующих декларативных операторах имена самых высоких уровней в структурах и строить цепочки наследования для каждой переменной отдельно.

Если переменная, участвующая в операции пересылки-присвоения, является именем переопределения или именем специального уровня, то определяются через соответствующие декларативные операторы имена переменных, которые переименовывают обрабатываемые переменные, и именно они заносятся в цепочку наследования.

Для облегчения работы аналитика, имена переопределения заносятся в таблицу алиасов модуля, в первой строке которой помещается исходное имя переменной, а далее следуют все имена, использованные в программе, которые тем или иным способом переименовывают исходное имя переменной. Таблица алиасов выдается аналитику вместе с анализируемым модулем для использования при прослеживании продвижения бизнес-термина внутри модуля.

Заключение. Цель статьи – обоснование предлагаемого подхода к реинжинирингу программных *legacy*-систем, а также описание алгоритмов, относящихся к начальному этапу реинжиниринга и ориентированных на автоматизацию построения БП-правил на стадии *Preprocessing*.

Получены следующие результаты:

- разработан механизм фиксирования и отслеживания состояния логических переменных с целью организации последовательности вызова БП. Для этого определены правила и методы построения статических и динамических векторов состояния для текущего отслеживания условий выполнения БП;
- предложена методика построения матрицы условий запуска БП, используемая, с одной стороны, как карта контроля при написании БП, а с другой – как механизм определения момента активизации БП в процессе их функционирования;
- определены условия корректного построения комплекса БП и методика их проверки;
- намечены пути автоматизации реализации предлагаемого подхода путем разработки средств автоматизации:

– для просмотра БП с целью автоматического построения матрицы условий их запуска и векторов состояния;

– для автоматического отслеживания корректности комплекса создаваемых БП в процессе их написания;

– для автоматизации связи с предметной областью.

- разработаны типовые алгоритмы: *построения* дерева вызова модулей *legacy*-системы,

определения межмодульных связей, отслеживания связей *наследования* между переменными в рамках отдельных модулей.

1. *Реализация* реинжиниринга программных legacy-систем / А.В. Анисимов, В.В. Белодед, Н.Д. Пашковец и др. // УСиМ. – 2008. – № 6. – С. 4—48.
2. *Tom Debevoise* Business Process Management with a Business Rule Approach. (Business Knowledge Architects, 2005) ISBN 0-9769048-0-2.
3. *Barbara* & GOLDBERG, Larry. The Business Rule Revolution. Happy About. – 2006 October 9. – ISBN 1-60005-013-1.

Поступила 04.01.2009

Тел. для справок: (044) 259-0690, 526-4187 (Киев)
E-mail: *ava@unicyb.kiev.ua*, *vladimir.bilodid@gmail.com*,